Available online at www.sciencedirect.com

ScienceDirect

IERI Procedia 4 (2013) 155 – 167

Procedia IERI

www.elsevier.com/locate/procedia

2013 International Conference on Electronic Engineering and Computer Science

Context-Aware Mobile Patient Monitoring Framework Development: A Detailed Design

Mahmood Ghaleb Al-Bashayreh*, Nor Laily Hashim, Ola Taiseer Khorma

School of Computing, College of Arts and Sciences, University Utara Malaysia (UUM), Sintok, Kedah, 06010, Malaysia

Abstract

Recent advances in mobile and wireless sensor technologies have introduced new domain requirements that must be satisfied in designing a Context-Aware Mobile Patient Monitoring Framework (CMPMF) to develop Context-Aware Mobile Patient Monitoring Systems (CMPMS). Although there have been few studies that designed CMPMFs to develop CMPMS, they have severe deficiencies in considering the emerging domain requirements. To address this gap, a detailed design of CMPMF is presented based on Model Driven Architecture (MDA). The resulting Platform Independent Model (PIM), Platform Specific Model (PSM), and code show that the detailed design has satisfied the domain requirements of CMPMF to develop CMPMS.

Keywords: Context-Aware Mobile Patient Monitoring Framework (CMPMF); Context-Aware Mobile Patient Monitoring Systems (CMPMS); Model Driven Architecture (MDA); application framework development; detailed design

1. Introduction

The application framework is a semi-complete application [1]. It provides a set of essential functionalities, which application developers must tailor and extend to build complete applications [2]. The process of

* Corresponding author. Tel.: +6-017-506-9168.
E-mail address: Mahmood.G.Al-Bashayreh@ieee.org.

doi:10.1016/j.ieri.2013.11.023

extending a framework is called framework instantiation and each resulted complete application, which customizes the framework, is called a framework instance [3].

A framework consists of concrete and abstract classes or interfaces [4], which are arranged into frozen spots and hot spots [5]. Frozen spots are concrete classes that are shared among all of the applications that are built using the framework [6]. These spots do not change (i.e., frozen), even when a framework is instantiated by applications [5]. Hot spots are abstract classes or interfaces, which represent framework flexibility and extensibility points that must be instantiated by application developers [7]. Hot spots are designed to be extended to meet application-specific needs [2]. A well-designed framework depends on the adequacy of its provided hot spots [5]. The methods in hot spots are called hook methods [6].

Framework instantiation is accomplished through hook methods. Hooks are the places in a framework where application developers can add their own code by extending the framework to meet an application-specific functionality. Framework developers define hooks as means to enable application developers to use and extend frameworks to build various applications for a specific domain [1].

Application framework extensibility, supported by hook methods within hot spots, is the dominant quality attribute that must be satisfied when developing frameworks [8]. A framework is considered useful if it is extensible [9]. Achieving extensibility ensures that a framework can be reused to develop domain-specific applications [7]. Framework extensibility techniques range from white-box to black-box [10] based on the hooks instantiation methods [5].

Framework development consists of six main activities: domain analysis, architectural design, framework design, framework implementation, framework testing, and documentation [11]. First, domain analysis is intended to explain the domain knowledge that is targeted by the framework [10], then capture the domain requirements from literature, domain experts, or existing standards for the domain [12]. This activity takes a long period of improvement. Thus, modeling domain knowledge is considered an ideal approach to reduce the duration of this activity [13]. Therefore, the main deliverable of this activity is a domain model [10], which includes the domain requirements and the relations among them [12].

Second, architectural design uses a domain model from previous activity as input to select the appropriate architectural style, forming the foundation of the framework. The selected architectural style results the main deliverable of this activity, which is the framework architectural design [11].

Third, a framework design uses the architectural design from previous activity as an input to be improved. Additionally, new classes are designed and different design patterns are applied to the framework. The main deliverables of this activity are the functionality scope given by the framework design, the framework's reuse interface, design rules based on architectural decisions that must be obeyed, and a design history document describing the design problems encountered and the solutions selected, with an argument for each [11].

Fourth, framework implementation focuses on coding both abstract and concrete classes [11]. Normally, object-oriented languages, such as C#, Java or C++, are used for implementing frameworks [14].

Fifth, framework testing is intended to identify if the framework satisfies the required functionality and to evaluate the framework reusability [11] and extensibility [8]. Reusability and extensibility are among the main characteristics that distinguish successful application frameworks [10]. Framework reusability, which is normally concerned with code and design [15], can be evaluated by instantiating the framework to develop sample applications [16]. Alternatively, framework extensibility can be evaluated through hooks [17].

Sixth, framework documentation includes documents that describe the purpose of the framework, the use of the framework, a user manual, and the design of the framework. Moreover, good documentation contains various examples including sample code for customizing and extending the framework [7]. In addition, design patterns can be used as a documentation approach to capture the framework design and help developers understand the framework [18].

This paper is a presentation of the third and fourth framework development activities, which are framework design and implementation, as part of ongoing research on designing a Context-Aware Mobile Patient Monitoring Framework (CMPMF) to develop Context-Aware Mobile Patient Monitoring Systems (CMPMS). In our previous work, the first activity, domain analysis, was conducted. A systematic review of 20 designed frameworks in the biomedical informatics domain was conducted. The results show that there are few studies that designed CMPMFs to develop CMPMS. Although there have been few studies that designed CMPMFs, they have a severe lack of any consideration of the emerging domain requirements. Additionally, no study integrates all of the identified requirements. Therefore, there is a need to address these emerging domain requirements of CMPMS in the design of CMPMFs [19; 20]. The second activity, architectural design, also was conducted and the results show that the internal architecture of the CMPMF layers and its components, as well as how they are structured, satisfy the requirements of CMPMS [21]. Therefore, the objective of this paper is to present a detailed design of CMPMF to address the identified domain requirements.

The paper is organized as follows. The proposed research framework to satisfy the objective of this paper is introduced in Section 2. Section 3 is a presentation of the framework design and implementation of the proposed CMPMF. Finally, section 4 is a presentation of the conclusion and a brief discussion of future work.

2. Research framework

To achieve the objective of this paper, the Model Driven Architecture (MDA) was adopted as a standard approach for Model-driven development (MDD) methodology to design and implement the proposed CMPMF. The suitability of the MDA approach to be applied for designing and implementing health care systems in a biomedical informatics domain was approved [22]. The MDA approach consists of three essential development activities. These activities are: analysis, low-level design, and coding. The outcome of the first process is a high-level abstract model, which is independent of any implementation technology; hence, it is called a Platform Independent Model (PIM). The outcome of the second process is a specific model, which is platform dependent; hence, it is called a Platform Specific Model (PSM) [23]. In fact, it is worth mentioning that each PIM can be transformed to one or more PSM according to the needs of the enterprise [24]. For example, a particular PIM can be transformed to a PSM in J2EE and a PSM in Microsoft .NET technology [25]. In addition, the PIM represents a conceptual model, while the PSM represents a physical model [22]. The outcome of the third process is a code model, which defines the code used for development [23]. These essential processes were adopted and customized in this research as PIM development, PSM development, and code development respectively. To facilitate applying the MDA approach, the Enterprise Architect tool (http://www.sparxsystems.com) was used, because it supports an automated transformation of the PIM to PSM and from PSM to code using built-in transformation rules. The Enterprise Architect tool also supports UML 2.3 based modeling. Additionally, it can generate code in C++, C#, and Java. Moreover, the Enterprise Architect tool can generate test methods for each public method. The following subsections elaborate on these three essential development activities.

2.1. Activity 1: platform independent model development

To construct the PIM, there are two common steps applied in the literature [22]. First, a UML class diagram should be constructed based on the identified requirements. Second, the class diagram should be refined by using four common techniques: hot spots, frozen spots, design patterns [26], and design principles [27]. These techniques are required to meet a number of the identified requirements [19; 20], which are: the

component-based development approach; the black-box framework extensibility approach; and the extensibility and reusability evaluation approaches.

2.2. Activity 2: platform specific model development

To construct the PSM, the resulting PIM should be transformed to PSM using a C# model transformation. This transformation takes the resulting PIM as an input and generates a PSM using special C# stereotypes. The C# platform technology was selected to support the following identified requirements. First, it supports component-based development [28]. Second, it supports the black-box extensibility approach using object composition [29]. Third, it supports soft real-time systems [30] required to support real-time continuous monitoring. Fourth, it supports asynchronous method calls [31] required to support a non-blocking communication with an unlimited number of sensors and unlimited number of mobile monitoring applications. Last, it supports cross-platform mobile development [32]. In fact, Microsoft .Net does not officially support any mobile platform except Windows. However, with the emergence of the Mono project (http://www.mono-project.com/), C# can be used to develop mobile applications that can be executed on various platforms other than the Microsoft Windows Phone, including iOS and Android [32]. Table 1 shows that C# is the best for writing native applications across various mobile platforms in comparison with other programming languages.

Table 1. Native mobile platform languages (adopted from [33])

	iOS	Android	Windows Phone
C/C++	✓	✓	
Objective-C	✓		
Java		✓	
Visual Basic.Net			✓
C#	✓	✓	✓

2.3. Activity 3: code development

To develop the code, the resulting PSM should be transformed to C# code using an automated tool [22]. Then some manual implementation should be developed. By comparison, the PIM development process is the only process that requires complete manual and innovative development. However, the PSM development process typically is automated [34], while the code development process is partially automated in this research.

3. Framework design and implementation

The architectural design of CMPMF is divided into two layers: the context monitoring layer and the context characterization layer. The context monitoring layer consists of five components: the data source collector, the data source connector, the data converter, the context information type, and the context monitoring manager. The context characterization layer consists of four components: the query element, the context monitoring query, the monitoring query evaluator and context characterization manager. This architectural design was discussed in detail in [21]. The architectural model resulting from the architectural design activity was used as an input to the framework design and implementation process. As discussed in Section 2, there are three essential activities to design and implement the CMPMF based on the MDA approach. The following subsections are an implementation of these activities.

3.1. Platform independent model development

Fig 1. shows the constructed PIM resulting from applying the PIM development activity. The four refinement techniques, mentioned in Subsection 2.1, are discussed in the following subsections.

3.1.1. Hot spots and frozen spots

First, hot spots represent the variable domain requirements that were identified in the domain analysis process. These variable domain requirements may be mapped to interfaces in the framework design. In fact, an interface-based design provides flexibility, reusability, and supports component-based development [10]. Second, frozen spots represent the common domain requirements. These common domain requirements may be mapped to concrete classes in the framework design. The identified interfaces and concrete classes of CMPMF are introduced as follows.

- ISubject interface: It provides the standard mechanism for communication between the framework components. It allows a particular component (e.g., component A) to request data from another component (e.g., component B) asynchronously. It also allows component B to notify component A once its requested data is ready. CMPMF provides the Event class as a default implementation of this interface.
- IObserver interface: It provides a callback method used to receive the event notifications of the ISubject.
- IDataValue interface: It is used as a general data type to store the collected data. It provides an extensibility point to represent various data values. CMPMF provides three concrete classes as default implementation of the IDataValue interface, which are: (1) DataValueScalar to represent data with a single value; (2) DataValueMinMax to represent data with minimum and maximum values; and (3) DataValueSet to represent data with a set of values.
- IDataSourceCollector interface: It provides the methods and properties required to collect data asynchronously from context data sources and convert the collected data if required from one format to another. IDataSourceCollector delegates these two responsibilities to IDataSourceConnector and IDataConverter respectively. IDataSourceCollector provides an extensibility point to represent various context data sources. CMPMF provides three concrete classes as default implementation of the IDataSourceCollector, which are: (1) DynamicDSC to collect data from dynamic data sources such as wireless sensors, (2) SemiDynamicDSC to collect data from semi-dynamic data sources such as mobile graphical user interface, and (3) StaticDSC to collect data from static data sources such as a mobile patient profile. The IDataSourceCollector observes the IDataSourceConnector and publishes an event notification once new data are collected. This is achieved through realizing the IObserver interface and composing the Event class respectively. This interface uses the IDataSourceConnectorFactory interface to create data source connectors using concrete class factories as a default implementation of the IDataSourceConnectorFactory.
- IDataSourceConnector interface: It provides the methods and properties required to connect asynchronously to the context data sources. It provides an extensibility point to represent various connection techniques that must be implemented by application developers. IDataSourceConnector observes the context data sources and publishes an event notification once new data are received. This is achieved through realizing the IObserver interface and composing the Event class respectively.
- IConnectionSettings interface: It is an empty interface responsible for encapsulating connection settings to connect to a specific context data source.
- IDataConverter interface: It provides a method that may be used to convert data from one format to another suitable format. It provides an extensibility point to represent various conversion algorithms that must be implemented by application developers.

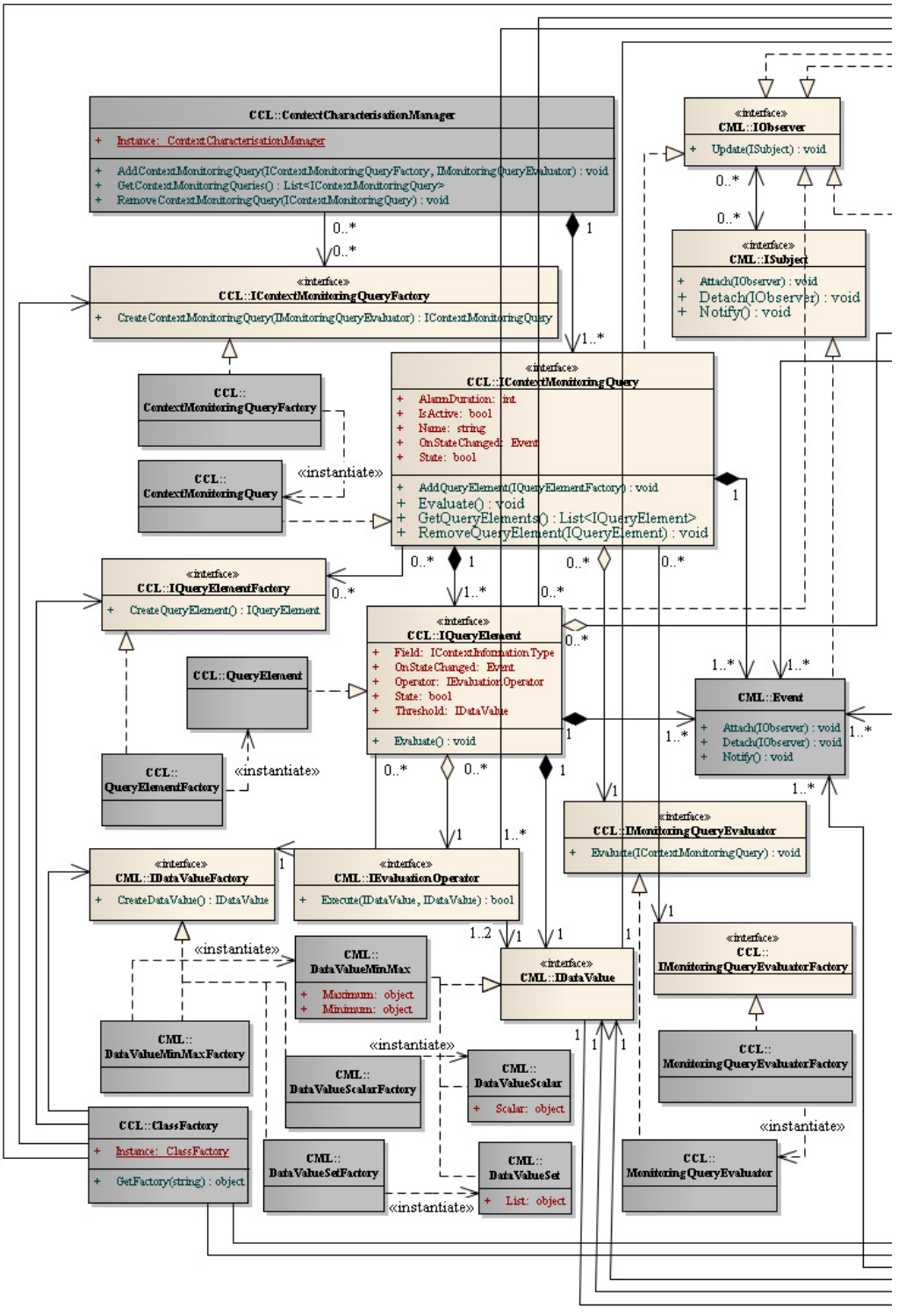

Fig. 1. CAMPMF platform independent model (part 1/2)

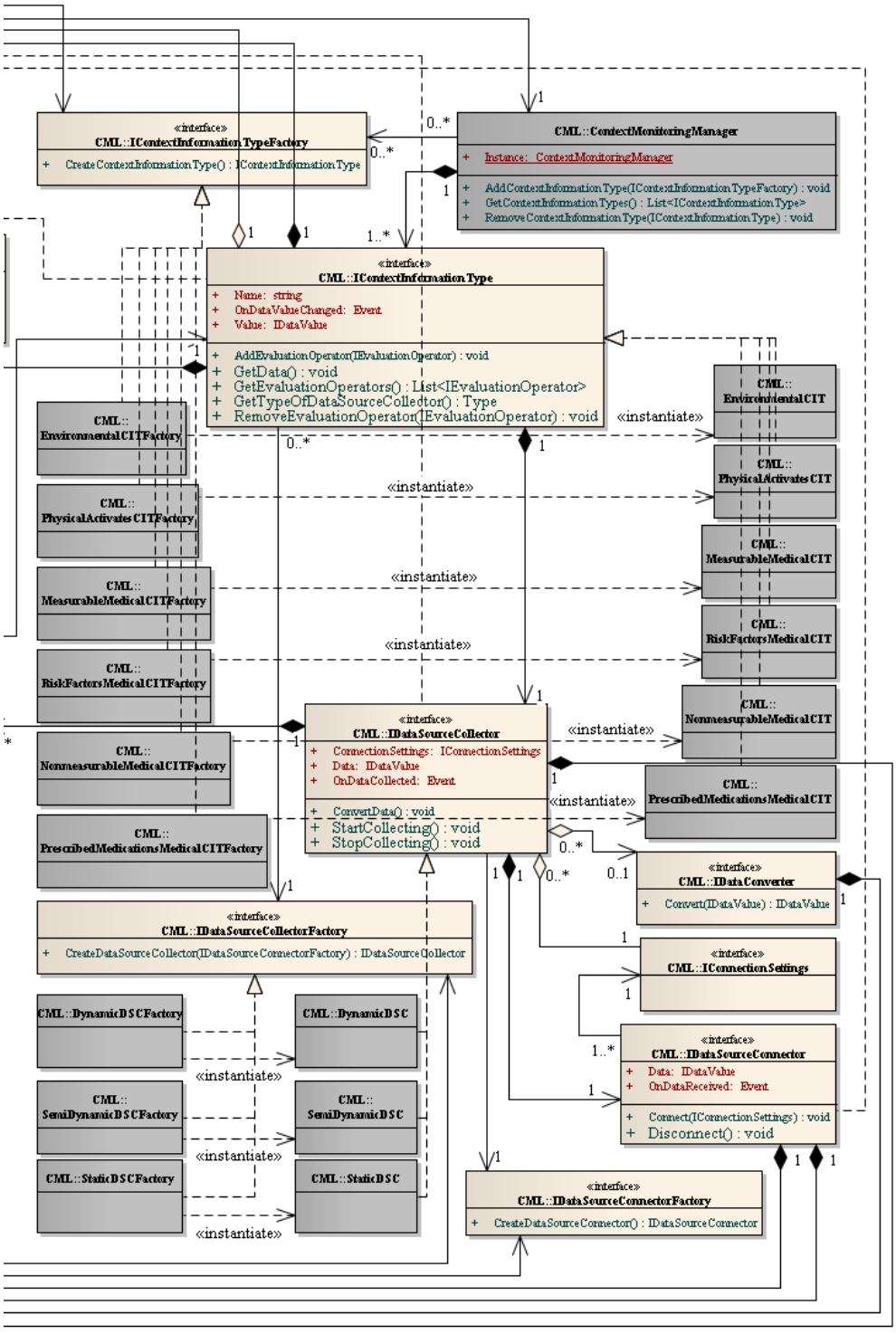

Fig. 1. Continued (part 2/2)

• IContextInformationType interface: It provides the methods and properties required to obtain context data asynchronously from context data sources. IContextInformationType delegates this responsibility to IDataSourceCollector. IContextInformationType provides an extensibility point to represent various context information types. CMPMF provides six concrete classes as a default implementation of the IContextInformationType, which are: (1) MeasurableMedicalCIT to represent a patient's vital signs (e.g., body temperature); (2) NonmeasurableMedicalCIT to represent medical symptoms (e.g., dizziness); (3) RiskFactorsMedicalCIT (e.g., cholesterol level); (4) PrescribedMedicationsMedicalCIT; (5) PhysicalActivatesCIT (e.g., sleeping); and (6) EnvironmentalCIT (e.g., room temperature). IContextInformationType observes the IDataSourceCollector and publishes an event notification once a data value is changed. This is achieved through realizing the IObserver interface and composing the Event class respectively. IContextInformationType aggregates a number of evaluation operators as an implementation of the IEvaluationOperator interface used to compare the data value of IContextInformationType with the threshold data value of the IQueryElement interface. This interface uses the IDataSourceCollectorFactory interface to create a number of data source collectors such as creating DynamicDSC using concrete class factories as a default implementation of the IDataSourceCollectorFactory.

• IEvaluationOperator interface: It provides a method required to execute a particular comparison operator (e.g., IsEqual or IsGreaterThan) to compare between the data value and threshold data value, which is used to evaluate a logical expression of a query element. IEvaluationOperator provides an extensibility point to represent various comparison operators that must be implemented by application developers.

• ContextMonitoringManager class: It is a concrete class that provides the methods required to manage context information types. It represents the primary interface of the context monitoring layer by providing a single access point to enable the context characterization layer to register to context information types. This class uses the IContextInformationTypeFactory interface to create a number of context information types such as creating MeasurableMedicalCIT using concrete class factories as a default implementation of the IContextInformationTypeFactory.

• IQueryElement interface: It provides the methods and properties required to create and evaluate a logical expression. IQueryElement delegates the responsibility of evaluating its logical expression to IEvaluationOperator. CMPMF provides a QueryElement class as a default implementation of this interface. IQueryElement observes the IContextInformationType and publishes an event notification once its state is changed based on the evaluation result. This is achieved through realizing the IObserver interface and composing the Event class respectively. This interface uses the IDataValueFactory interface to create a number of data values such as DataValueScalar using concrete class factories as a default implementation of the IDataValueFactory.

• IContextMonitoringQuery interface: It provides the methods and properties required to create and evaluate a context monitoring query to characterize a patient medical situation based on the state of IQueryElement. IContextMonitoringQuery delegates the responsibility of evaluating the context monitoring query to IMonitoringQueryEvaluator. CMPMF provides a ContextMonitoringQuery class as a default implementation of this interface. IContextMonitoringQuery observes the IQueryElement and publishes an event notification once its state is changed based on the evaluation result. This is achieved through realizing the IObserver interface and composing the Event class respectively. This interface uses the IQueryElementFactory interface to create a number of query elements. It also uses the IMonitoringQueryEvaluatorFactory interface to create a number of monitoring query evaluators. This creation is accomplished using concrete class factories as a default implementation of both IQueryElementFactory and IMonitoringQueryEvaluatorFactory respectively.

• IMonitoringQueryEvaluator interface: It provides a method used to evaluate a context monitoring query to characterize a patient medical situation based on the state of IQueryElement and the types of context data

sources. It provides an extensibility point to represent various evaluation algorithms that must be implemented by application developers. CMPMF provides a MonitoringQueryEvaluator class as a default implementation of this interface.

- ContextCharacterizationManager class: It is a concrete class that provides the methods required to manage context monitoring queries. It represents the primary interface of the context characterization layer by providing a single access point to enable the CMPMS to register to context monitoring queries. This class uses the IContextMonitoringQueryFactory interface to create a number of context monitoring queries using concrete class factories as a default implementation of IContextMonitoringQueryFactory.
- ClassFactory: It is a concrete class that provides a method used to load the concrete class factories and return concrete objects.

3.1.2. Design patterns and design principles

First, design patterns are defined by Gamma et al. [18] as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." Design patterns describe proven design ideas and knowledge. Thus, they are very important to be used by designers [35]. Therefore, it is recommended to use as many patterns as possible in designing and applying application frameworks [10]. A framework can realize or instantiate one or more patterns into an executable artifact [36]. Additionally, patterns can be used as a method to document application frameworks, because they provide a common vocabulary for depicting software design to help developers to understand the framework [18]. Second, design principles are good ideas help software developers to build better design. Design patterns are used as tools for applying the design principles. There are five primary design principles support reusability and extensibility [27], which are:

- Single Responsibility Principle (SRP), which ensures that "a class should have only one reason to change" [27];
- Open-Closed Principle (OCP), which ensures that "software entities (e.g., classes, modules, functions) should be open for extension, but closed for modification" [27];
- Liskov Substitution Principle (LSP), which ensures that "subtypes must be substitutable for their base types" [27];
- Dependency Inversion Principle (DIP), which ensures that "(1) high-level modules should not depend on low-level modules. Both should depend on abstractions. (2) Abstractions should not depend on details. Details should depend on abstractions" [27]; and
- Interface-Segregation Principle (ISP), which ensures that "Clients should not be forced to depend on methods that they do not use" [27].

In this research, the design patterns: singleton, observer, strategy, and abstract factory were used as strategies for applying the above mentioned design principles. These design patterns and design principles were applied to refine the PIM as discussed in the following subsections.

3.1.2.1. Singleton design pattern

In the proposed design of CMPMF, only one instance of the ContextMonitoringManager class and the ContextCharacterisationManager class must be instantiated to provide a global access point to their layers, which are the context monitoring layer and context characterization layer respectively. To consider the limited resources of mobile devices, these two classes should not be initialized until invoked for the first time, which is known as lazy instantiation. To meet this need, the singleton design pattern [18] that offers lazy instantiation [37] was used.

3.1.2.2. Observer design pattern

In the proposed design of CMPMF, an asynchronous communication between the framework components is used as a standard mechanism for communication. This allows a particular component (e.g., component A)

to request data from another component (e.g., component B) asynchronously. It also allows component B to notify component A by raising an event once its requested data are ready. Such a mechanism is required to support the communication between the following four pairs of components: (1) IDataSourceConnector and IDataSourceCollector; (2) IDataSourceCollector and IContextInformationType; (3) IContextInformationType and IQueryElement; (4) IQueryElement and IContextMonitoringQuery. To meet this need, the observer design pattern was used [18; 37]. This pattern ensures non-blocking communication and supports connecting to an unlimited number of context data sources and notifying an unlimited number of CMPMS. Moreover, this pattern conforms to the OCP that allows registering new observers (e.g., CMPMS) without changing the subject (e.g., IContextMonitoringQuery). Looking back to Fig. 1, it can be seen that the IContextInformationType, IContextMonitoringQuery, IDataSourceCollector, IDataSourceConnector, and IQueryElement are substitutable for the IObserver and the Event concrete class is substitutable for the ISubject. Therefore, the LSP is applied. Furthermore, the Event concrete class depends on the IObserver interface and the concrete methods of the ISubject also depends on the IObserver interface. Thus, the DIP is applied.

3.1.2.3. Strategy design pattern

In the proposed design of CMPMF, there is a need to enable, for example, the IDataSourceCollector to collect data from various context data sources, including dynamic context data sources (e.g., wireless body sensors), semi-dynamic context data sources (e.g., a mobile graphical user interface), and a static context data sources (e.g., a mobile patient profile). Each of these context data sources requires different connection techniques and each connection technique may require different connection settings. Such need requires defining different communication strategies or algorithms and encapsulating them, so that any of these strategies or algorithms can interchange with each other to support the extensibility of the proposed CMPMF. This situation appears between the following components: (1) IDataSourceCollector and its IDataSourceConnector as well as IDataConverter; (2) IContextInformationType and its IDataSourceCollector; (3) IQueryElement and its IEvaluationOperator; and (4) IContextMonitoringQuery and its IMonitoringQueryEvaluator. To meet this need, the strategy design pattern was used [18; 37]. This pattern conforms to the DIP that allows each concrete class to be manipulated by various algorithms (strategies)[27]. It also fully conforms to the OCP that supports the component-based development and black-box extensibility approach using the composition approach.

3.1.2.4. Abstract factory design pattern

In the proposed design of CMPMF, the SRP was applied by decoupling responsibilities across different components to support CMPMF extensibility. In addition, the DIP was applied by depending on interfaces rather than concrete classes. For example: (1) the IDataSourceCollector delegates the responsibility of collecting data asynchronously from various context data sources to IDataSourceConnector; (2) the IContextInformationType delegates the responsibility of obtaining context data asynchronously from context data sources to IDataSourceCollector; (3) IContextMonitoringQuery delegates the responsibility of evaluating the context monitoring query to IMonitoringQueryEvaluator. However, to connect these components, there is a need to provide a mechanism to prepare an object from a server component (e.g., DynamicDSC concrete class) to serve a client component (e.g., MeasurableMedicalCIT concrete class). This object preparation responsibility must be delegated to an intermediary component. To meet this need, the abstract factory design pattern was used [18; 37]. This pattern supports component-based development and the black-box extensibility approach using the composition approach.

3.2. Platform specific model development

In this research, as discussed in Section 2, PIM was transformed using a C# model transformation into PSM by using an automated tool.

3.3. Code development

In this research, as discussed in Section 2, PSM was transformed to code using a C# code transformation template by using an automated tool, in addition to some manual code development, see Fig 2.

```
namespace CaMPaMF.CML.Core
{
    // Hook
    public interface IDataSourceConnector
    {
        IDataValue Data {get; set; }
        // Hook method.
        void Connect(IConnectionSettings connectionSettings);
        // Hook method.
        void Disconnect();
        //Manual code development
        event EventNotificationHandler<IDataSourceConnector> OnDataReceived;
    }
}
```

Fig. 2. Sample automated implementation of the interface IDataSourceConnector.

4. Conclusion and future work

This paper is a presentation for a detailed design of ongoing research about designing a CMPMF to develop CMPMS. It begins with an introduction to application framework development including their six main development activities. Then, the need for a detailed design of CMPMF is highlighted. It also includes the research framework that was used based on the MDA approach to satisfy the research objective of this paper. The detailed design and implementation of CMPMF including PIM, PSM, and code development were presented. The results show how the domain requirements of CMPMF have been satisfied by the proposed detailed design. This resulting detailed design can be used by researchers to design enhanced CMPMF to develop CMPMS. In the future, the researchers will attempt to develop three CMPMS as case studies using CMPMF to evaluate framework reusability and extensibility.

References

[1] Al-Dallal J. Testing object-oriented framework applications using FIST2 tool: a case study. *Int J Electr Comput Syst Eng* 2010;4:119-126.
[2] Oliveira ALS, Alencar P, and Cowan D. ReuseTool—An extensible tool support for object-oriented framework reuse. *J Syst Software* 2011;84:2234-2252.
[3] Gurp Jv, and Bosch J. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Software Pract Ex* 2001;31:277-300.
[4] Markiewicz ME, and Lucena CJPd. Object oriented framework development. *Object oriented framework development* 2001;7:3-9.
[5] Pree W. Essential Framework Design Patterns. *Object Mag* 1997;7:34-37.
[6] Schmid HA. Systematic framework design by generalization. *Comm ACM* 1997;40:48 - 51.

[7] Mili H, Fayad M, Brugali D, Hamu D, and Dori D. Enterprise frameworks: issues and research directions. *Software Pract Ex* 2002;32:801-831.
[8] Driver C, and Clarke S. An application framework for mobile, context-aware trails. *Pervasive and Mobile Computing* 2008;4:719-736.
[9] Fayad M, Hamu DS, and Brugali D. Enterprise frameworks characteristics, criteria, and challenges. *Comm ACM* 2000;43:39-46.
[10] Fayad M, Schmidt DC, and Johnson RE. Application Frameworks. In: Fayad M, Schmidt DC, and Johnson RE, (editors). *Building Application Frameworks: Object-Oriented foundations of Framework Design*, New York, NY: Wiley; 1999, p. 3-28.
[11] Bosch J, Molin P, Mattsson M, Bengtsson P, and Fayad M. Framework Problems and Experiences. In: Fayad M, Schmidt DC, and Johnson RE, (editors). *Building Application Frameworks: Object-Oriented foundations of Framework Design* New York, NY: Wiley; 1999, p. 55-82.
[12] Arango G. Domain analysis method. In: Schäfer W, Prieto-Díaz R, and Matsumoto M, (editors). *Software Reusability*, New York, NY: Ellis Horwood; 1994, p. 17-49.
[13] Aksit M, Marcelloni F, and Tekinerdogan B. Developing object-oriented framworks using domain models. *ACM Comput Surv* 2000;32:11.
[14] Cunningham HC, Liu Y, and Zhang C. Using classic problems to teach Java framework design. *Sci Comput Program* 2006;59:147-169.
[15] Al-Dallal J. Estimating the coverage of the framework application reusable cluster-based test cases. *Inform Software Tech* 2008;50:595-604.
[16] Binder R. *Testing Object-Oriented Systems: Models, Patterns, and Tools (ARP/AOD)*. Reading, MA: Addison-Wesley; 1999.
[17] Al-Dallal J. Testing object-oriented framework hook methods. *Kuwait J Sci Eng* 2008;35:103-122.
[18] Gamma E, Helm R, Johnson R, and Vlissides JM. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley; 1995.
[19] Al-Bashayreh MG, Hashim NL, and Khorma OT. Context-Aware Mobile Patient Monitoring Frameworks: A Systematic Review and Research Agenda. *J Software* 2013;In Press.
[20] Al-Bashayreh MG, Hashim NL, and Khorma OT. The Requirements to Enhance the Design of Context-Aware Mobile Patient Monitoring Systems Using Wireless Sensors. In: Vinh PC, Hung NM, Tung NT, and Suzuki J, (editors). *Context-Aware Systems and Applications*, Berlin, Germany: Springer; 2013, p. 62-71.
[21] Al-Bashayreh MG, Hashim NL, and Khorma OT. Context-Aware Mobile Patient Monitoring Framework Development: An Architectural Design. *Adv Sci Lett* 2013;In Press.
[22] Raghupathi W, and Umar A. Exploring an MDA approach to health care information systems development. *Int J Med Informat* 2008;77:305-314.
[23] Kleppe A, Warmer J, and Bast W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA: Addison Wesley; 2003.
[24] Hailpern B, and Tarr P. Model-driven development: The good, the bad, and the ugly. *IBM SYST J* 2006;45:451-461.
[25] Gorton I. *Essential Software Architecture* Berlin, Germany: Springer 2006.
[26] Chen X. *Developing application frameworks in .NET*. Berkeley, CA: Apress; 2004.
[27] Martin RC. *Agile software development: principles, patterns, and practices*. Upper Saddle River, NJ: Prentice Hall; 2003.
[28] Szyperski C. *Component Software: Beyond Object-Oriented Programming*. 2nd. New York, NY: Addison-Wesley; 2011.
[29] Cooper JW. *C# Design Patterns: A Tutorial* Boston, MA: Addison-Wesley; 2003.
[30] Lutz MH, and Laplante PA. C# and the .NET framework: ready for real time? *IEEE Software* 2003;20:74-80.
[31] Davies A. *Async in C# 5.0*. Sebastopol, CA: O'Reilly; 2012.

[32] Olson S, Hunter J, Horgen B, and Goers K. *Professional Cross-Platform Mobile Development in C#.* Indianapolis, IN: Wiley; 2012.
[33] Shackles G. *Mobile Development with C#.* Sebastopol, CA: O'Reilly; 2012.
[34] Jones VM, Halteren Av, Konstantas D, Widya I, and Bults R. An application of augmented MDA for the extended healthcare enterprise. *Int J Bus Process Integrat Manag* 2007;2:215–229.
[35] Crnkovic I, Hnich B, Jonsson T, and Kiziltan Z. Basic Concepts in CBSE. In: Crnkovic I, and Larsson M, (editors). *Building Reliable Component-Based Software Systems*, Norwood, MA: Artech House; 2002, p. 3-22.
[36] Zhang W, and Kim M. What Works and What Does Not: an Analysis of Application Frameworks Technology. *JBSGE* 2006;1:15-26.
[37] Bishop J. *C# 3.0 Design Patterns*. Sebastopol, CA: O'Reilly; 2008.